049051-223

ENCAPSULATED OBJECT ORIENTED POLYPHASE PREBOOT EXECUTION AND SPECIFICATION LANGUAGE

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention relates generally to specifying and executing computing tasks in a preboot execution environment, and specifically to an encapsulated object oriented polyphase preboot execution and specification language.

Description of the Related Art

Network-based computing has been gaining popularity in recent years as a more desirable approach to managing enterprise computing needs. Ranging from network-managed PCs, network computers, and thin-clients, network-based computing is largely motivated by the need to reduce the cost of providing IT services (known as the Total Cost of Ownership, TCO) in networked computing environments. It is well known in the industry that the most expensive part of providing computing resources to end-users is not the cost of the computing hardware and software but the cost of on-going maintenance and management. See, Thin Client Benefits, Newburn Consulting (2002); Total Cost of Application Ownership, The Tolly Group Whitepaper (1999); TCO Analyst: A White Paper on GartnerGroup's Next Generation Total Cost of Ownership Methodology, GartnerConsulting (1997). According to the well known studies, network-based computing approach dramatically reduces the TCO by centralizing the maintenance and management functions of IT services, thereby reducing the recurring cost of on-going maintenance and management.

A key challenge in network-based computing is specifying client dependent behaviors or parameters so that various types of client devices can be centrally managed effectively. For example, for Remote Imaging – that is, installing images on client device from centrally managed image servers – it is crucial to ascertain and maintain accurate information about client hardware in order for the system to work effectively. As well known to those skilled in the art, many tools currently exist to manage client device configurations in a networked computing environment. However, nearly all of such tools operate *after* the client device has completed its bootstrap operation – i.e., after an operating system has been loaded on the client. But, some of the most critical tasks in centralized network device management take place before the boot – that is, during what is called as the preboot environment. For example, remote imaging operation is usually initiated during a preboot phase. It is desirable to transfer operating system images to the client computer before it boots so that the client computer boots with the newly installed operating system images or components. Clearly, this preboot phase is the most critical aspect of remote imaging, as the client computer will not function properly at all if wrong operating system image or components were delivered. Ideally, what is needed is a tool that runs at the client computers during the preboot because the client machine is the best place to ascertain what hardware it has and what operating system and software it needs. Furthermore, a maximum flexibility would be afforded by a generalized specification language for specifying and executing computing tasks during the preboot phase. For example, although preboot imaging and operating system installation is a complex process, much of the process can be parameterized at some abstract level. For instance, installation of device drivers, often the most challenging part of installation, can be parameterized as: if X device is found, install Y device driver file. Moreover, overall scheme of operation of remote imaging may be quite similar for many types of devices, e.g., downloading bootstrap code, OS kernel, device drivers,

system DLLs, etc.. Thus, such operation can be parameterized if the unknown parameters are resolved at the client side by the client devices during execution of remote imaging operations. However, no vendor currently provides a generalized specification language for the preboot execution environment.

Without a generalized specification language for the preboot execution environment, the specification and management of preboot computing tasks at the central server is a highly complex process with many inherent risks. The device type of the clients must be ascertained beforehand, and images and components must be prepared for each device types within which all devices must have identical hardware. Then, the client device type must be verified when the images and components are delivered to the client devices. The complexities and difficulties of this process is mainly attributable to information mismatch – that is, trying to manage at the server side operations which critically depends on information at the client side. Thus, a generalized specification language for the preboot execution environment which runs at the client machines will also greatly simplify the management functions at the central server. Instead of complex scripts and programs to prepare images and components, all that is needed is a simple specification that provides: if x, y, z hardware or configuration is found, perform X, Y, Z tasks.

It can be seen, then, there is a need in the field for a method for specifying and executing computing tasks in a preboot execution environment.

SUMMARY OF THE INVENTION

Accordingly, the present invention addresses the foregoing need by providing method for specifying and executing computing tasks in a preboot execution environment in general, and by providing, in particular, an encapsulated object oriented preboot execution and specification language.

According to one aspect of the invention, the present invention is an encapsulated object-oriented polyphase language for specifying computing tasks in multiple phases of generating and executing preboot execution specification, comprising: a computing task specification generator; and a computing task interpreter, wherein generated computing task specifications are encapsulations, encapsulating execution environment dependent parameters, and wherein the generated computing task specifications are polymorphic with respect to the encapsulated parameters, as well as to the multiple phases of generating and executing preboot execution specification.

The multiple phases of generating and executing preboot execution specification can be: a definition phase, wherein computing tasks are defined; a generating phase, wherein specifications for the computing tasks are generated; and an execution phase, wherein the specifications for the computing tasks are executed.

According to another aspect of the invention, the present invention is an encapsulated object-oriented polyphase language, wherein the behavior of language itself is polymorphic with respect to the multiple phases of generating and executing preboot execution specification.

Other and further objects and advantages of the present invention will be further understood and appreciated by those skilled in the art by reference to the following specification, claims, and drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Referring now to the drawings in which like reference numbers represent corresponding parts throughout:

FIG. 1 illustrates an overview of a system according to the present invention;

FIG. 2 illustrates a preboot execution environment according to the present invention;

FIG. 3 illustrates downloading and operation of a language agent at a client computer according to the present invention;

FIG. 4 illustrates a logical view of downloading a language agent according to the present invention;

FIG. 5 illustrates downloading a specification file according to the present invention;

FIG. 6 illustrates downloading a series of specification files according to the present invention;

FIG. 7 illustrates a conceptual overview of logical operation of an encapsulated object-oriented polyphase language according to the present invention;

FIG. 8a to FIG. 8c show example code for generating a polyphase encapsulation according to the present invention; and

FIG. 9 illustrates a polyphase encapsulation generated by the example code shown in FIG. 8a to FIG. 8c according to the present invention.

DETAILED DESCRIPTION OF THE INVENTION

FIG. 1 illustrates an overview of a system according to the present invention. As shown in FIG. 1, the system environment in which the present invention operates comprises Image Server (110), Server Language Agent (112), Specification and Image (114) generated by Server Language Agent (112), Client Computer (120), Client Language Agent (122), and Network (130). As also shown in FIG. 1, Specification and Image (114) generated by Server Language Agent (112) is delivered to Client Computer (120) over Network (130), and processed by Client Language Agent (122) to perform computing tasks specified in Specification and Image (114).

In one aspect of the invention, the present invention is a system for executing computing tasks in a preboot execution environment, comprising a language agent with a preboot execution language interpreter. As well known to those skilled in the art, a preboot execution environment is a computing environment at a computer before the computer completes booting – i.e., completes loading an operating system (OS).

FIG. 2 illustrates a preboot execution environment according to the present invention. As shown in FIG. 2, a preferred embodiment of the present invention operates in the Preboot Execution Environment (PXE). PXE is a widely-adopted industry standard for network booting or remote booting over a network that provides a way for network cards to initiate a network connection to servers before any OS is loaded so that the OS images or components can be downloaded over the network. See, "Preboot Execution Environment (PXE) Specification Version 2.1", by Intel Corporation, September 1999. As illustrated in FIG. 2, ImageDisk/BootDisk (210) is downloaded from Image Server (110) to Client Computer (120) over Network (130) utilizing PXE facilities. ImageDisk/BootDisk (210) contains computer code that initializes operation of Client Computer (120) and provides Execution Environment (220) in Program Memory (230) of Client Computer (120). A preferred embodiment of Execution Environment (220) is Wyse Imaging System (WISard) execution environment. However, Execution Environment (220) can be any preboot execution environment known to those skilled in the art without departing from the scope of the present invention.

To briefly describe the operation of network boot and PXE, the booting process of a client computer starts at the ROM BIOS (240) of Client Computer (120) which contains code for recognizing the network interface card (NIC) as an IPL Device (Initial Program Load Device) from which to boot and load an operating system. See, "BIOS Boot Specification", by Compaq Computer Corporation, Phoenix Technologies Ltd., and Intel Corporation, January 1996.

The network card in turn must also be a bootable device such as a PXE-enabled NIC. PXE includes DHCP (Dynamic Host Configuration Protocol) that allows IP address assignment to the NIC at Client Computer (120) so that Client Computer (120) may communicate with Image Server (110) over Network (130) utilizing industry standard TCP/IP network protocol. In addition, PXE supports TFTP (Trivial File Transfer Protocol) of TCP/IP suite for transferring files over Network (130). The network card can also employ any preboot communication protocol known to those skilled in the art such as the IBM RPL (Remote Program Load) without departing from the scope of the present invention.

When Client Computer (120) initiates booting, the BIOS Boot code instructs the PXE-enabled NIC to perform PXE operation – DHCP, DHCP-proxy and TFTP transfer – to obtain the initial OS boot code, which in turn connects to a boot server, for example, Image Server (110), to download the initial OS boot code, which can be provided by an image, DiskImage (210), or a proprietary archive, BootDisk (210). When loaded to Program Memory (230) of Client Computer (120), DiskImage/BootDisk (210) code provides Execution Environment (220) such as WISard. For legacy OS support, Execution Environment (220) then traps the pre-OS disk access requests (INT 13 in PC architecture) and redirects them to the PXE-enabled NIC so that the necessary OS files can continued to be downloaded to Client Computer (120). For non-legacy OS support, INT13-interception is not required since bootstrap may contain all intelligence to perform preboot functions. That is, no INT13 related translation is required since the bootstrap contains operating intelligence (such as WISard). In either case, the entire boot process is completely transparent to users on Client Computer (120).

FIG. 3 illustrates downloading and operation of a language agent at a client computer according to the present invention. As shown in FIG. 3, the first image file downloaded to Client Computer (120), designated as Root.i2u (310),

contains code for Client Language Agent (122). All subsequent image files downloaded are processed by Client Language Agent (122).

A key component of Client Language Agent (122) is a preboot execution language interpreter, which is a general purpose interpreter which can interpret and execute specifications for preboot computing tasks in accordance with the supported syntax and semantics. The preboot execution language interpreter of Client Language Agent (122) of the present invention, thus allows processing of any computing task at Client Computer (120) by transmitting to Client Computer (120) files which contain specifications for preboot computing tasks. Since a computer language is by nature symbolic, abstract entity, this process is better understood when viewed from a symbolic, or more accurately, a logical perspective.

FIG. 4 illustrates a logical view of downloading a language agent according to the present invention. Any file or image is an Endpoint (410), which in the present case is Root.i2u (310). A more complete definition and description of an Endpoint is given below. As shown in FIG. 4, Root.i2u (310) provides code that operates as Client Language Agent (122) in WISard Execution Environment (220).

FIG. 5 illustrates downloading a specification file according to the present invention. As shown in FIG. 5, a specification file, designated as Rule.i2d (510) is downloaded from Image Server (110), and processed by Client Language Agent (122). The preboot execution language interpreter of Client Language Agent (122) interprets the specifications for executing computing tasks contained in Rule.i2d (510) and performs the specified computing tasks. Rule.i2d (510) can also contain instructions to download further specification files.

FIG. 6 illustrates downloading a series of specification files according to the present invention. As shown in FIG. 6, a series of specification files, designated as Rule0.i2d (610), Rule1.i2d (612), Rule2.i2d (614), and RuleN.i2d (616), are downloaded in succession by chain download instructions contained in

the files. Since this process can be continued for an arbitrary number of times, any computing task can be specified and performed in the preboot execution environment. The computing tasks can be those for general imaging, platform imaging, remote imaging, remote booting, preboot computer diagnostics, and computer preparations ("prepping") such as formatting and partitioning the hard disks on Client Computer (120), without departing from the scope of the present invention.

The veracity of these propositions, remarkable they may be, would become evident when the syntax of the preboot execution and specification language of the present invention is described. First, definitions of the terms are given. (1) "Method" is the means to provide data through an endpoint. Some examples of methods are stream/, fswyse/, fsfat/, or any other WISard supported method. (2) "Location" provides for a complete description of a physical interface; this is where data is actually located. Examples of location are: ide0-0/, ide0-1/, ide1-0, and ide1-1/ for IDE hard disk drives; nand/ and nor/ for flash memory devices; or any WISard supported location or device. (3) "Reference" is used as an operator to define a sub-part of a method/location/definition. An example of a reference name is a file on a device through a file-system, e.g., fsfat/ide1-0/file, or any WISard supported reference. (4) "Endpoint" is a Location of a data image, Method of retrieval, and Reference name. Endpoints are generally of the form: Method/Location/Reference. (5) "Pipe" is a pair of well-defined endpoints comprising source and destination information from which to read and write data.

Given the foregoing definitions, the syntax of the preboot execution specification language of the present invention is as follows. First, the preboot execution specification language of the present invention provides an instance copy language where a left-side (LHS) is copied to the right hand-side (RHS). It operates in a manner similar to a familiar MS-DOS 'copy' operation: "LangPXE f1 f2", where the instance f1 is copied and a new instance f2 is created by the

language agent LangPXE. Note that f1 and f2 may also contain any additional drive or path information formally utilized by the file-system rules. Second, the language of the present invention provides a means of device copying describing a physical path to the instance: "LangPXE physical-path/f1 physical-path/f1". An example of this type of copying is copying entire content of one hard disk drive and place it onto another hard disk. For instance, when "LangPXE ide0-0/f1 ide0-1/f2" is carried out, a duplicate of ide0-0 drive is created on ide0-1 drive.

Third, the language of the present invention provides a means of copying or applying an endpoint to the instance: "LangPXE Method1/Location1/Reference1 Method2/Location2/Reference2." For example, "LangPXE stream/ide1-0/disk.raw tftp/disk.raw" would copy disk ide1-0 to file disk.raw over a network using TFTP. Fourth, the language of the present invention provides a means of describing a Transport for the purpose of executing its described copy operations. In other words, the Transport contains instructions for executing copy operations. A Transport is also called an Archive in the present invention. Also, it is said that the copy operations are "encapsulated" in the Transport or Archive. For example, "LangPXE stream/ide1-0/disk.raw archive/rule.i2d tftp/disk.raw" would encapsulate the copy operation "stream/ide1-0/disk.raw tftp/disk.raw" in the archive/rule.i2d file. In general, the form of the syntax for this purpose is: "LangPXE LHS ARCHIVE RHS". Fifth, the language of the present invention provides a means of describing the extraction (or unpacking) of an archive for the purpose of performing an encapsulated copy operation. For example, "LangPXE archive/rule.i2d" will result in execution of operations encapsulated in archive/rule.i2d.

Sixth, the language of the present invention provides a means of defining symbols for LHS, RHS, and ARCHIVE type. For example, "LangPXE f1 definesymbols/" will define symbols specifically defined in the instance

represented by f1. Seventh, the language of the present invention provides symbolic or parametric LHS, RHS, and ARCHIVE types. The parametric types are denoted by < > notation. For example, "LangPXE <f1> archive/<T1> <f2>" encapsulates operations "LangePXE <f1> <f2>" into Transport archive/<T1>, where <f1> and <f2> can substitute for any allowable LHS and RHS forms, respectively, and <T1> can be any Transport reference. In other words, <f1>, <T1>, and <f2> are "variables" or "parameters". <T1> and <f2> can be NULL, i.e., nonexistent.

Eighth, the language of the present invention provides de-referencing of LHS. De-reference operation is denoted by ~ operator. For example, with "LangPXE ~f1 archive/T1 f2", archive/T1 created will not contain the instance represented by f1 until archive/T1 is actually executed. Instead, archive/T1 will contain "~f1 f2", which specifies "f1 f2" operation when archive/T1 is executed. In contrast, with "LangPXE ~f1 archive/T1 f2", archive/T1 created contains "f1 f2" operation, which is executed when archive/T1 is executed. Thus, ~ operator effects de-referencing. Another way to understand de-referencing is "nesting" of operations at successive phases of operations specified. "f1 f2" runs the specified operation at the level (or phase) of archive/T1 execution. In contrast, "~f1 f2" runs "f1 f2" operation at the level of execution of one of the tasks specified by execution of archive/T1, that is, one level down or nested from archive/T1. That is, the object represented by f1 is not contained in the archive/T1, but only the reference is specified. When archive/T1 is executed, f1's object is satisfied and the object is copied to f2. Ninth, the language of the present invention provides de-referencing to the Nth degree, that is de-referencing nested to the Nth level. For example, "LangPXE ~~f1 archive/T1 f2" will create the instance of f1 only after the second level event specified in the Transport archive/T1 takes place.

Tenth, the language of the present invention provides a means of initiating and terminating encapsulation (or encoding) of archives (or transports). "LangPXE ~encode archive/T1 T2" will result in all subsequently defined LHS

to be placed or referenced in T1 to be placed into T2, and "LangPXE ~encodeoff archive/T1 T2" will terminate the encoding or encapsulation process for T2.

Eleventh, the language of the present invention provides a means of referencing other archives from a single archive. "LangPXE archive/t2 archive/t1" will result in archive/t2 called from archive/t1. Twelfth, the language of the present invention provides a means of returning when called from another archive. "LangPXE RETURN archive/t1" will result in returning to the calling archive which called archive/t1.

Thirteenth, the language of the present invention provides a means of describing conditional operations. "LangPXE ~if archive/t2" will result in the evaluation of the next completed pipe description and process all pipes immediately following the evaluation if the process returned success. If however, it returned false, the process would skip all arguments contained in the transport until coming upon the syntax, "LangPXE ~endif archive/t2."

Finally, the language of the present invention provides a means of encapsulating within an encapsulation, denoted by @ operator. For example, "LangPXE @archive/t2 archive/t1" will encapsulate archive/t2 within archive/t1, so that when archive/t1 is executed one of the tasks performed is execution of archive/t2 archive.

Although the language of the present invention is primarily directed to a preboot execution environment, it should also be noted that the language is a general language that can be operated in any execution environment known to those skilled in the art without departing from the scope of the present invention.

Now, with the syntax described above, it is possible to specify and execute any preboot computing tasks, including general imaging, platform imaging, remote imaging, remote booting, preboot diagnostics, and preboot prepping. For example, a "pull" operation in imaging, where images are copied by a client from a server, can be simply specified as: "stream/ide1-0/disk.raw ftp/disk.raw", which will result in making an duplicate copy of ide1-0 disk over a network by

using FTP. Furthermore, "Platform.k stream/sec/validate; stream/ide1-0/disk.raw ftp/disk.raw" will perform: (1) validating Platform is of correct type; and (2) in making a duplicate copy of ide1-0 disk over a network by using FTP. Similarly, a simple "push" operation, where images are copied by a server to a client, can be specified as: "ftp/disk.raw stream/ide1-0/disk.raw". Remote booting can be accomplished in an analogous fashion, since remote booting is simply providing images necessary over a network to boot a client computer.

Client or Target specific customization can be accomplished by utilizing the parametric capability of the language of the present invention. For example, "<GET00> <T00> <PUT00>", when "<GET00>=ValidLHS," and "<T00>=NULL," will provide an immediate pull-and-push operation, where the object represented by LHS is written as an object to RHS. This is a pull-and-push operation (Immediate Write). Using the same example, "<GET00>=ValidLHS, and <T00>=archive/ValidTransport," will provide any generalized object-pipe operation where object represented by ValidLHS syntax is immediately placed into an encoding named <T00>. This operation is a pull operation (Immediate Read) which, when <T00> is later executed by the language interpreter, will result in a completed copy operation that places the data-object now contained in the transport <T00> to the now valid RHS reference <PUT00>. This secondary operation is a push operation (Transported Write). In contrast to this example, "<GET00> <T00> <PUT00>", where "<GET00>=~ValidLHS" and "<T00>=archive/ValidTransport," will provide any generalized reference-pipe operation, where ValidLHS reference-syntax is placed into an encoding named <T00>, which, when <T00> is later executed by the language interpreter, will result in read from LHS (ValidLHS) and a write to RHS reference <PUT00>. This is a delayed-pull-and-push operation (Delayed Immediate Write).

This differs from the first generalized statement when the current executed transport itself has syntax to encode yet another transport by use of the "~encode" operation. In this example, "<GET00> <T00> <PUT00>" where

"<T00>=archive/ValidTransport" and "<GET00>=+ValidLHS," causes the object read during the encoding process to be placed into the newly created encoding (future transport). This is an Encoded-Immediate-Read operation and will result in a literal object being placed into a transport as did the previously described pull operation encoding above. Note that in all cases shown, the operators delayed when the existence of LHS occurred and where the placement of the eventual object was to be placed (RHS or Transport).

The parametric specification and execution capability of the language of the present invention affords one aspect of object-oriented character of the language. For instance, "<method1>/<location1>/disk1.raw <method2>/<location2>/disk2.raw" is a general specification for a pull-and-push operation. Furthermore, the archive that contains this line is an encapsulation of generalized pull-and-push operation. While an encoding specifying "<method1>/<location1>/disk1.raw archive/<T00> <method2>/<location2>/disk2.raw" is a generalized pull operation, placing all drive contents into the transported archive indicated by the symbol <T00>; later executing the archive <T00> will result in a push operation.

As illustrated above, the language of the present invention has capability for symbolic translation, that is, denoting parameters with symbols and resolving that an appropriate time. Moreover, the symbol definitions can be placed in a separate file, a symbol file, to translate or resolve the symbols. Another implication of symbolic translation is the ability to modify the encapsulations by replacing symbols with resolved definitions or with yet another symbol or symbols, since a symbol, by definition, can be anything.

In another aspect of the invention, the present invention is a system for specifying computing tasks in a preboot execution environment, comprising a language agent with a preboot execution specification generator. As described in the syntax specification above, the language of the present invention is an interpreter of preboot computing task specification as well as a generator of

preboot computing task specification. For example, the archives or transports generated by the present language are encapsulated specifications. Therefore, the language of the present invention is also employed by Server Language Agent (112) to generate preboot computing task specifications. The specified computing tasks can be any tasks supportable by the present invention, including, but not limited to, general imaging, platform imaging, remote imaging, remote booting, preboot diagnostics, and preboot prepping.

Thus, complex scripts generated by the existing technologies can be reduced to merely a few lines with parametric specifications. Furthermore, by utilizing parameters and symbols, target or client dependent parameters can by specified in encapsulations before even knowing what the target or client machines will be. With symbolic translation and capability to include encapsulations within another encapsulation, the present invention provides a powerful, flexible, generalized method of specifying computing tasks in preboot environment. Therefore, encapsulations can be used provide a generalized specification for any class of computing operations, including, but not limited to, general imaging, platform imaging, remote imaging, remote booting, preboot diagnostics, and preboot prepping. Such operations or tasks can be encapsulated in an encapsulation of the present invention even when the ultimate target destination or client environment is not known at the time of definition or specification.

From the foregoing, it is evident that the language of the present invention has at least two behaviors in three distinct phases – that is, the language behaves as a specification generator at the server during task definition phase, and as a specification interpreter and generator during the specification generation and execution phase. In the example discussed above, "<method>/<location>/disk.raw archive/<T00> <method2>/disk.raw", the distinct phases are: a definition phase, where the encoding is defined; a generation phase, where the archive <T00> is generated by pull operation; and an execution

phase, where push operation is performed when the archive <T00> is executed. However, there is no reason to limit specification function at the server or during the definition phase. Since the operations described by the present language can be de-referenced or nested to Nth degree by the ~ operator, execution of an archive can also generate specification by appropriate instructions. In addition, the @ operator allows encapsulating entire archives within an archive. Thus, the language of the present invention exhibits behavior that changes depending on the execution environment. In object-oriented design literature, this type of behavior is called Polymorphic behavior. Therefore, the language of the present invention has polymorphic behavior with respect to the different "Phases" of operation. That is, the present language is a "Polyphase" language.

According to another aspect of the invention, the present invention is an encapsulated object-oriented polyphase language for specifying computing tasks in multiple phases of generating and executing preboot execution specification, comprising: a computing task specification generator; and a computing task interpreter, wherein behaviors of generated specification are polymorphic with respect to the multiple phases of generating and executing preboot execution specification. The multiple phases of generating and executing preboot execution specification comprise: a definition phase, wherein computing tasks are defined; a generating phase, wherein specifications for the computing tasks are generated; and an execution phase, wherein the specifications for the computing tasks are executed.

FIG. 7 illustrates a conceptual overview of logical operation of an encapsulated object-oriented polyphase language according to the present invention. As shown in FIG. 7, a computing task in a preboot execution environment can be specified as a set of Pipes (710), comprising a pair of Endpoints (720 and 730), where Endpoint (720) as well as Endpoint (730) can be any source or target. A set of Pipes (710) is then executed by Client Language Agent (122) under WISard Execution Environment (220).

Because the language of the present invention is a polyphase language, exactly the same language is used for both Client Language Agent (122) and Server Language Agent (112). This means that both Client Language Agent (122) and Server Language Agent (112) can generate encapsulations *and* interpret (or execute) the encapsulation. Thus, an encapsulation of the present invention can be modified, propagated, multiplied, or otherwise manipulated in any way. For example, an encapsulation can translate symbols and modify itself or create another encapsulation by Server Language Agent (112) or Client Language Agent (122). Client Language Agent (122), while executing an encapsulation, can generate yet another encapsulation that is to be executed at a later time, e.g., when some desired parameters can be ascertained appropriately, such as additional hardware.

FIG. 8a to FIG. 8c show example code for generating a polyphase encapsulation according to the present invention; and FIG. 9 illustrates a polyphase encapsulation generated by the example code shown in FIG. 8a to FIG. 8c according to the present invention. FIG. 9 shows a view of the generated encapsulation, which is a binary image file, by using a binary file viewer. The right pane (910) shows an ASCII representation, and the left pane (920) shows a hexadecimal (Hex) representation.

The fact that the specification files, or rule encapsulations are binary files – that is, binary images – underscores an important point. It is that the archives, i.e., the rule or specification files, are simply a part of the image set for the overall imaging process. They are downloaded by PXE as if the files are just any OS component files. Thus, the images are self-describing images. In other words, the images contain all instructions necessary to execute or accomplish their purpose, such as installing the images, booting from the images, or executing any function or set of functions specified. Furthermore, the images encapsulate all parametric behaviors resolved during the appropriate levels of execution nesting.

Thus, all target customization is encoded in the image files themselves. The image files are encapsulated method-images.

Encapsulated imaging is particularly suited for Platform Imaging, which is defined as installing operating system images on a computing device. Examples of Platform Imaging are: installing Windows XP on a PC, installing Windows CE on an embedded computing device, and installing PocketPC on a hand-held device. In general, Platform Imaging is a target computing device dependent operation because what operating system can be installed on a target depends on the type and capabilities of the target hardware. Thus, Platform Imaging can be advantageously accomplished by encapsulated imaging, since all target dependent parameters can be encapsulated in the images themselves.

According to one aspect of the invention, the present invention is a system for encapsulated platform imaging, comprising a language agent with an encapsulated language interpreter for executing an encapsulation, wherein the encapsulation contains all instructions and data necessary to install an operating system onto a computing device. Imaging of an entire platform is accomplished through a single, self-describing encapsulation instance by executing or interpreting the self-describing encapsulation by the language agent with an encapsulated language interpreter. For this purpose, the encapsulation can be considered as persistent encapsulation of operating system class. The operating system can be any operating system known to those skilled in the art, including, but not limited to, Windows XP, Windows 2000, Windows NT, Windows Me, Windows 98, Windows CE, PocketPC, various flavors of Unix systems, and Linux, without departing from the scope of the present invention. The computing device can be any computing device known to those skilled in the art, including, but not limited to, a desktop PC, a notebook PC, an embedded computing device, and a hand-held device. It is important to note that, for platform imaging, the operation is not necessarily limited to preboot environments. For example, an update operation of an existing operating system installation need not take place

during preboot time. It can be performed while an operating system is up and running.

Encapsulated platform imaging of the present invention provides several advantages over existing methods. According to the present invention, only a single encapsulation is required to obtain a desired state on the client device. Since all target dependent information are parameterized and encapsulated, the same class-object definition can be used to accomplish platform imaging for any platform. Moreover, as described above, the language agent for executing an encapsulation itself can be a part of the method-image encapsulation. Hence, encapsulated platform image is self-contained as well as self-describing, and, therefore, an encapsulation is complete by itself to accomplish platform imaging. No programs or codes need to be preserved separate from the images.

Furthermore, all supporting or ancillary operations or computing tasks necessary to accomplish platform imaging can also be encapsulated in the encapsulated method-image itself. For example, hardware configuration and capabilities of a target device can be ascertained and compared for validation process. A status or result of one operation or process can be reported to another. Ancillary tasks such as hostname preservation, OS image preservation, and maintaining and configuring BIOS version and CMOS settings can be encapsulated in the same single encapsulation instance, representing the update operation of all of these parameters.

According to the present invention, an encapsulation and the language agent with an encapsulated language interpreter can be provided over a logical connection. A logical connection can be any method of providing data known to those skilled in the art, including, but not limited to, a computer readable medium and a network connection without departing from the scope of the present invention. A computer readable medium can be any computer readable medium known to those skilled in the art, including, but not limited to, a diskette, a compact disc, and a flash disk device without departing from the scope of the

present invention. Thus, platform imaging can be accomplished through a single encapsulation instance stored on or delivered from a diskette, a compact disc, a flash disk device, or a network connection.

In addition, a bootable interface, which is an initial boot facility, can be provided on or over a logical connection. For instance, a diskette or compact disc containing an encapsulation can be a bootable disk or a bootable compact disc. Similarly, a network connection can provide an initial boot facility by a bootable NIC such as a PXE-enable NIC. With a bootable interface provided, a target device can boot from the logical connection, load the language agent with an encapsulated language interpreter, load the rest of the encapsulation, and interpret or execute the encapsulation with the language agent to accomplish the entire platform imaging operation. Thus, the computer readable medium or a network connection endpoint represent a complete, self-contained, self-describing method of platform imaging. Encapsulated platform imaging with encapsulated method-image is simple to define, create, maintain, inventory, and distribute, reducing the complex process of platform imaging to a simple, elegant, and straightforward procedure.

It can seen, then, from the foregoing that the language of the present invention provides a more robust, reliable, and accurate execution or performance of specified tasks, as the target specific parameters are resolved only when the parameterized information becomes available at the appropriate phase to discover the pertinent information. The result is a robust, reliable, flexible, and simple method and system for centralized maintenance and management of client devices in a networked enterprise computing environment with a lower total cost of ownership than existing products.

The foregoing description of the preferred embodiment of the invention has been presented for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise form disclosed. Many modifications and variations are possible in light of the above teaching. It

is intended that the scope of the invention not be limited by this detailed description, but by the claims and the equivalents to the claims appended hereto.